Program Structures
Principles of straight-line code

- Make dependences obvious:

```plaintext
firstResult = doThing1();
secondResult = doThingY(firstResult);
```

- Vs.

```plaintext
doThing1();
doThingY();
```
Principles of straight-line code, cont.

- If no dependences, group related statements
Which is better?

A
```java
if (getAmountOfGasInTank() >= gasNeeded(destination)) {
    // avoid unnecessary stops; reduce wear on engine
} else {
    fillGasTank();
}
```

B
```java
if (getAmountOfGasInTank() < gasNeeded(destination)) {
    fillGasTank();
} else {
    // avoid unnecessary stops; reduce wear on engine
}
```

C  Control flow is fine for both

D  Control flow is problematic for both
Principles of if/else

- write the common case first; then write the unusual cases
- use boolean functions to simplify complicated tests

- Use case/switch only when it applies
  - Don’t do nasty things with it

```c
if (c == '*' || c == '/' || c == '+' || c == '-')
    //
    //
    //
    //

if (isOperator(c))
```
What is wrong with this code?

```java
/**
 * @param input a String
 * @return the input String with dashes and the characters following the dashes removed
 */
public String removeDashesAndFollowingChars(String input) {
    StringBuilder output = new StringBuilder();
    for (int i = 0; i < input.length(); i++) {
        char currentChar = input.charAt(i);
        if (currentChar != '-') {
            output.append(currentChar);
        } else {
            i++; // skipping character after dash
        }
    }
    return output.toString();
}
```
Which is better?

A

```java
public static Map<Integer, Integer> generateHistogram(int[] data) {
    Map<Integer, Integer> histogram = new HashMap<Integer, Integer>();
    for (int value : data) {
        int count = 1 +
        (histogram.containsKey(value) ? histogram.get(value) : 0);
        histogram.put(value, count);
    }
    return histogram;
}
```

B

```java
public static Map<Integer, Integer> generateHistogram(int[] data) {
    Map<Integer, Integer> histogram = new HashMap<Integer, Integer>();
    for (int i = 0; i < data.length; i++) {
        int value = data[i];
        int count = 1 +
        (histogram.containsKey(value) ? histogram.get(value) : 0);
        histogram.put(value, count);
    }
    return histogram;
}
```

C  Control flow is fine for both

D  Control flow is problematic for both
Which is better?

A

```java
public int[] copyIntArray(int[] input) {
    int[] copy = new int[input.length];
    int i = 0;
    for (int value : input) {
        copy[i++] = value;
    }
    return copy;
}
```

B

```java
public int[] copyIntArray(int[] input) {
    int[] copy = new int[input.length];
    for (int i = 0; i < input.length; i++) {
        copy[i] = input[i];
    }
    return copy;
}
```

C  Control flow is fine for both
D  Control flow is problematic for both
### Which is better?

**A**

```java
boolean dashFound = false;
for (String arg : args) {
    if (arg.equals("-")) {
        dashFound = true;
    } else if (!dashFound) {
        process1(arg);
    } else {
        process2(arg);
    }
}
```

**B**

```java
int i = 0;
while(i < args.length && !args[i].equals("-")) {
    process1(args[i]);
    i++;
}

i++; // skip the dash
for( ; i < args.length ; i++) {
    process2(args[i]);
}
```

**This code takes an array of strings, it processes all of the strings before a dash one way and all of the remaining strings another way**

**C** Control flow is fine for both

**D** Control flow is problematic for both
int dashPosition = args.length();

for (int i = 0; i < args.length(); i++)
{
    if (args[i].equals("-"))) 
    
        dashPosition = i;
        break;
}

for (int i = 0; i < dashPosition; i++)
    process1(args[i]);

for (int i = dashPosition+1; i < args.length(); i++)
    process2(args[i]);
Returns

- use early returns to reduce nest, eliminate cases
  - guard clauses
- minimize the number of returns in a routine
  - all things being equal

```java
boolean f(int value) {
    if (value < 0) {
        return false;
    }
    // ...
To Dos for Tuesday

- Read Ch. 19 (General Control Issues)
- Read Ch. 24 (Refactoring)

- Write an interactive ‘adventure game’ for your rooms.json

- Accepted commands:
  - done, exit, quit
  - go <direction>

- Handles all other inputs gracefully
  - detailed specification later today